PARALLEL CACHELETS

BACKGROUND

[1]   The present invention relates to an architecture for cache hierarchies in a computing system.

[2]   Contemporary processor designs often address bandwidth constraints imposed by system buses by providing on-chip or on-package cache memory. Various caching schemes are known.   Some schemes provide a single unified cache that stores both instruction data and variable data.   Other schemes provide multi-level cache hierarchies in which a lowest level cache, the cache closest to the processor core, is very fast but very small when measured against higher level caches. The higher level caches may be increasingly larger and, therefore, may be slower than their lower level counterparts, but their greater capacity tends to reduce data evictions rates and extend the useful life of cached data.   To improve hit rates and to avoid cache pollution, still other caching schemes may provide separate instruction and data caches, which enable higher hit rates due to increased code or data locality found in typical application's code and data streams.

[3]   Future microprocessors may employ superscalar designs in which multiple instructions will be executed in a single cycle. In this domain, a cache hierarchy that responds only to a single data request per cycle may become a bandwidth-limiting element within the processor. Although some known cache schemes, such as multi-ported or multi-banked caches, can accommodate multiple load requests in a single cycle, these known schemes are difficult to implement.   Multi-port architectures are complex and difficult to build as separate circuitry is required to accommodate each additional port. Multi-banked architectures introduce complex circuitry to recognize and manage bank conflicts.   Accordingly, the inventors perceive a need in the art for a low-complexity cache hierarchy in which a single level of cache may respond efficiently to multiple load requests per cycle.

BRIEF DESCRIPTION OF THE DRAWINGS

[4]   FIG. 1 is a block diagram of a parallel cache design according to an embodiment of the present invention.

[5]   FIG. 2 illustrates a format of data exchange that may occur between requesters and the cache, according to an embodiment of the present invention.

[6]   FIG. 3 illustrates a method of operation suitable for use in an embodiment that permits a dynamic assignment of cachelets to new requests.

[7]   FIG. 4 is a block diagram of a data processing system according to an embodiment of the present invention.

[8]   FIG. 5 is a block diagram of a data processing system 400 according to an alternate embodiment of the present invention.

[9]   FIG. 6 is a block diagram of a cachelet 500 according to an embodiment of the present invention.

[10]   FIG. 7 illustrates multi-port simulation and implicit set associativity properties the may be achieved by a parallel cachelet architecture.

[11]   FIG. 8 illustrates a cache hierarchy of an agent according to an embodiment of the present invention.

DETAILED DESCRIPTION

[12]   Embodiments of the present invention provide parallel cachelets in a microprocessor. A single level of cache may include a plurality of independently addressable cachelets. The level of cache may accept multiple load requests in a single cycle and apply each to a respective cachelet. Depending upon the content stored in each cachelet, the cachelet may generate a hit/miss response to the respective load request. Load requests that hit their cachelets may be satisfied therefrom. Load requests that miss their cachelets may be referred to another level of cache.

[13]   FIG. 1 is a block diagram of a parallel cache 100 according to an embodiment of the present invention. The parallel cache 100 may include a plurality of cachelets 110-1 through 110-N, each storing data for use by other elements of an agent in which it sits. Each cachelet (say, 110-1) may be addressed independently of each other cachelet. Thus, the cachelet 110-1 may receive an address signal (addr1) on an input that is independent from each other input. The cachelet 110-1 may generate a hit/miss response indicating whether the requested data is present within the cachelet and, if the data is present, may output the data from the cache in a response.

[14]   FIG. 1 illustrates the cache 100 integrated into a larger system. Plural load requests may be issued by one or more requesters 120-1 through 120-M. Although the four requesters are illustrated in the example of FIG. 1 along with four cachelets, these numbers are coincidental. The number of cachelets 110-1 to 110-N may but need not be the same as the number of requesters 120 provided in the agent.

[15]   FIG. 1 also illustrates an interconnection fabric 130 providing communication between the requesters 120 and the cachelets 110-1 to 110-N. The interconnection fabric manages load requests issued from the requesters 120 and forwards load requests to each of the cachelets 110-1 to 110-N. In an embodiment, the interconnection fabric may be provided within the cache 100. In other embodiments, the interconnection fabric may be external to the cache 100. In another embodiment (not shown in FIG. 1), the interconnection fabric may be omitted altogether.

[16]   The cache design of FIG. 1 advantageously provides a cache in which multiple load requests may be satisfied in a single cycle. Each load request may be applied to a respective cachelet 110.1-110.N. If the load hits the cachelet, the data may be supplied to the requester. If not, the load may be referred to another level of cache (not shown) or to system memory (also not shown).

[17]   FIG. 2 illustrates a format of data exchange that may occur between requesters and the cache, according to an embodiment of the present invention. As shown in FIG. 2, communication may occur according to a request 200 and reply 210 communication protocol. Such protocols are well-known and widely used in modern processing agents. Conventionally, a request 200 would include a load request identifying, for example, a

request type identifying the uses to which the requester will put the requested data (e.g., read, read for ownership, etc.) and an address of the requested data (fields not shown).

[18]   Requests 200, according to an embodiment, may include a field 220 for conventional load data and additionally may include a valid field 230 and a cachelet pointer ("CP") field 240. In an embodiment, the cachelet pointer 240 may identify a cachelet within the cache to which the request 200 is directed. The valid field 230 may identify whether data in the cachelet pointer 240 is valid.

[19]   When a reply by the cache 100 furnishes data 250 in response to a request. The reply may identify a cachelet from which the data was furnished. Thus, replies from the cache 100 also may include a cachelet pointer 260 identifying the cachelet that furnished the data.

[20]   FIG. 3 illustrates a method of cachelet assignment according to an embodiment of the present invention. The method may begin be determining whether the load request is addressed to a cachelet (box 1010). This determination may be made by checking the valid bit of the request. If not, a cachelet assignment may be made according to a default assignment scheme (box 1020). For example, a cachelet may be assigned to the load request according to a round-robin or least-recently-used assignment scheme. Thereafter, or if the load request was addressed to a cachelet, the method may determine whether the load request conflicts with another load request (box 1030). If two load requests conflict, if they both are addressed to the same cachelet, then one of the load requests is assigned to an unused cachelet (represented by box 1040).

[21]   Once all load requests are assigned to a cachelet, a lookup may be performed at each cachelet. Each cachelet may determine whether its respective load request hits the cachelet (box 1050). If so, the requested data may be read from the cachelet and furnished to the requester (box 1060). If not, the load request may advance to other layers of cache, possibly to system memory, until the requested data is found and stored in the cachelet (box 1070). Thereafter, the requested data is furnished to the requester (box 1060). When data is returned to the requesters, it may include a cachelet pointer identifying the cachelet that furnished the data.

[22]   By way of example, consider operation of the method 1000 when a plurality of loads L_1-L_{12} are received for the first time. Each of the loads may be assigned to a respective cachelet according to the default scheme. When data is returned in response to the loads, a CP pointer may identify the cachelet assignment that was made for the respective load. New loads may continue to be assigned cachelets according to a round-robin scheme. Thus, in this example, loads L_1-L_{12} after having been delivered to the cache 100 for the first time, may have been given cachelet assignments as shown below in Table 1.

LOAD	VALID BIT	CACHELET POINTER	CACHELET ASSIGNMENT
L_1	Invalid	---	0
L_2	Invalid	---	1
L_3	Invalid	---	2
L_4	Invalid	---	3
L_5	Invalid	---	0
L_6	Invalid	---	1
L_7	Invalid	---	2
L_8	Invalid	---	3
L_9	Invalid	---	0
L_{10}	Invalid	---	1
L_{11}	Invalid	---	2
L_{12}	Invalid	---	3

Table 1

Table 1 also illustrates contents of the valid bit and cachelet pointer of the loads L_1-L_{12} as they might be presented to the cache.

[23]   Of course, due to the dynamic behavior of programs, the grouping of loads can be expected to vary significantly.   There can be no expectation that new loads will be presented to a cache simultaneously or that a grouping of loads that were presented collectively to the cache in one cycle will be presented as a group in some later cycle.

[24]   Consider an example wherein loads L_3, L_4, L_7 and a new load, L_{13}, are presented to the cache 100 in a single cycle. The content of the valid bit and the cachelet pointer of these load is illustrated in Table 2.

LOAD	VALID BIT	CACHELET POINTER	CACHELET ASSIGNMENT
L_3	Valid	2	2
L_4	Valid	3	3
L_7	Valid	2	1
L_{13}	Invalid	---	0

Table 2

Loads L_3 and L_7 present a conflict; both are addressed to cachelet 2. Load L_{13} is new, it may be assigned to the next cachelet in the round-robin scheme, cachelet 0. To resolve the conflict between loads L_3 and L_7, one may be presented to cachelet 2; the other may be assigned to an unused cachelet. In the example shown in Table 2, load L_7 is shown as being assigned to cachelet 1. Thereafter, each load may be applied to its respective cachelet to determine whether requested data is present therein. If so, the cachelet may furnish the requested data to the requester and identify the cachelet in which the requested data was found.

[25]   In the example of Table 2, load L_7 is shown being redirected from cachelet 2 to cachelet 1. Although a load may be redirected, the load need not always miss the new cachelet. It is possible, due to processing of other loads, that the data sought by load L_7 will be present in the cache. Embodiments of the present invention permit multiple copies of the same data to be present in more than one cachelet.

[26]   Redirection of loads in the presence of a cachelet conflict can improve bandwidth for load requests. Without such redirection, given the conflict shown in Table 2, only one of the loads L_3 or L_7 could be granted. If load L_3 were granted, load L_7 would have to be stalled until the address cachelet completed processing for load L_3. Redirection permits the load L_7 to be granted also, even in the face of the cachelet conflict. With redirection, the load L_7 may be applied and unused cachelet and, if the cachelet stores the requested data, the load may be satisfied earlier than if it were stalled.

[27]   FIG. 4 is a block diagram of a data processing system according to an embodiment of the present invention. The system 300 may include an instruction decoder 310, one or more load/store units 320 and a cache 330. The decoder 310

decodes program instructions. It may pass load instructions to the load/store units 320 for execution. In this regard, the operation of the system 300 is well known.

[28]   The cache 330 may include cachelets 340 as described above and an address manager 350. The address manager 350 may perform cachelet assignment as described herein. In the embodiment shown in FIG. 4, cachelet assignment may occur at execution. As a load instruction is being executed, the load/store units pass load requests to the cache. The address manager 350 may receive the load requests and perform cachelet assignments as the requests are being received.

[29]   FIG. 5 is a block diagram of an alternate embodiment of a data processing system 400 according to the present invention. This embodiment 400 also may include an instruction decoder 420, one or more load/store units 420 and a cache 430. In this embodiment, however, each of the load/store units 420 may be mapped to only one of the cachelets 440 within the cache.

[30]   In this embodiment, a cachelet manager may be provided in a communication path between the instruction decoder 410 and the load/store units 420. Cachelet assignment functions may occur as load instructions are passed to the load/store units 420.

[31]   Providing cachelet assignment functions at instruction execution as in FIG. 4 allows dynamic information to be used than when cachelet assignment functions are performed at instruction decoding. When instruction execution is performed out of order, instruction may be executed as soon as execution resources and data dependencies are resolved. Instead of assigning cachelet to loads that are concurrently decoded, the embodiment of FIG. 4 performs cachelet assignments on concurrently executing loads. Simulation studies suggest that the embodiment of FIG. 4 may outperform the embodiment of FIG. 5. Alternatively, however, the embodiment of FIG. 5 removes an interconnection network between the load/store units and the cachelets. The embodiment of FIG. 5, therefore, benefits from a simpler implementation. Because the address manager is not part of the load execution, the embodiment of FIG. 5 may reduce cache access latency and possibly cycle time.

[32]   FIG. 6 is a block diagram of a cachelet 500 according to an embodiment of the present invention. The cachelet 500 may include a plurality of cache lines 510-514. Each cache line may include a set field 520 and a tag field 530 each of a predetermined length. During operation, within a particular set, the tag field 520 may store a "tag" representing a portion of the data's address. The data field 530 may store the data itself.

[33]   The cachelet 500 may include an address decoder 540. The address decoder may index one of the sets 510-214 in response to an address signal applied on an input 550 to the cachelet 500. During a write of data to the cachelet 500, the address decoder activates one of the cache lines (say, 511) identified by the address signal, causing data to be stored in the cache line 511. During a read of data from the cachelet 500, activation of the cache line 511 may the cache line 511 to output stored data from the cachelet 500. Typically, only a portion of a complete address is used to activate a cache line 511; FIG. 1 labels this data Addr$_{set}$. According to convention, the cache line 511 activated in response to the address signal is called a "set."

[34]   The cachelet 500 also may include a tag comparator 550, having two inputs. A first input is provided in communication with the tag field 530 of each of the cache lines 511-214. A second input receives a portion of the address signal, called "Addr$_{tag}$" in FIG. 1. During a read operation, when a new address signal is applied to the cachelet 500, the set information Addr$_{set}$ will activate one of the cache lines 511 and cause tag data therefrom to be output to the tag comparator 550. Thus, the tag comparator 550 may compare an externally supplied tag with tag data stored in an addressed cache line. When the two tags agree, the tag comparator 550 may generate an output identifying a tag hit; otherwise, the tag comparator may generate an output indicating a miss.

[35]   The cachelet 500 also may include a gate 560 coupled to the output of the cache lines 511-214 and controlled by the tag comparator 550. The gate 560 may be rendered conductive in response to a cache hit but non-conductive in response to a cache miss. Thus, the gate 560 may permit the cachelet 500 to output data only when requested data is present in the cachelet 500.

[36]   The cachelet structure may be repeated for each of the cachelets in a layer of cache. Structures for multiple cachelets 500, 570 and 580 are illustrated in FIG. 6.

[37]   The cachelet structure illustrated in FIG. 6 shares much in common with traditional set associative caches. Conventional set associative caches typically include a set of cache banks arranged in ways. All ways in set associative caches, however, are addressed by the same address during cache lookups. They cannot process multiple loads in a single clock cycle. By contrast, each of the cachelets 500, 570 580 may be addressed independently of each other and, therefore, have the potential to process multiple loads per cycle.

[38]   An interesting aspect of the behavior of the parallel cachelets is that they provide a form of implicit set associativity. Since the same entry of all the cachelets can hold different data, an implicit form of set associativity may be constructed within a cache. Even if the cachelets are direct mapped, if two loads with addresses that would normally be direct mapped to the same entry of a cache actually are assigned different cachelets, then both loads can be satisfied. This effectively allows the parallel cachelets to behave as a set associative cache, with the same entry of all the cachelets forming a set. The number of cachelets determines the maximum number of "ways" of the implicit set associativity. However, this is not quite the same as normal set associativity in that there is less flexibility in the use of the entries of a set. Also, the replacement policy within the set is determined by the cachelet assignment process; effectively, it may become random and not LRU as is found in traditional set associative caches.

[39]   Complementing implicit set associativity, the parallel cachelet scheme also provides a form of on-demand replication that can facilitate concurrent memory accesses. Replication allows multiple dynamic loads to access the same data concurrently in multiple cachelets. The replication also decreases the potential for cachelet conflicts. It is possible, however, that too much replication can become wasteful and can reduce cachelet hit rates.

[40]   Hence, the parallel cachelet scheme provides two important attributes: on-demand replication of data in multiple cachelets to provide increased bandwidth (simulating multi-porting); and implicit set associativity that allows different data to be

stored in the same entry of different cachelets to provide better overall hit rate (simulating set associativity).

[41]   FIG. 7 provides an example illustrating these two attributes. Assume all the cache schemes employ direct-mapped addressing, and that addresses A, B, and C all map to the same cache set. Assume that in a particular machine cycle i, there are four dynamic loads with addresses: A, A, B, and C. Both the multi-banked and multi-ported schemes will only be able to provide one data element due to their direct mapped addressing. The parallel cachelet scheme has the potential to support accessing to all three addresses, A, B, and C, if they are assigned to different cachelets (implicit set associativity). Furthermore, the parallel cachelet scheme permits multiple copies of the same data element in A to exist within the cache (on-demand replication). Hence the parallel cachelet scheme has memory accessing flexibility that can adapt to dynamic program behavior. The parallel cachelet scheme can actually outperform a direct-mapped multi-ported cache for some benchmarks due to implicit set associativity and on-demand replication.

[42]   FIG. 8 illustrates a cache hierarchy 600 of an agent according to an embodiment of the present invention. The cache hierarchy 600 shares much in common with conventional cache hierarchies: multiple levels 610, 620 and 630 of cache (labeled Layer 0-Layer 2) may be provided between an agent's core (not shown) and system memory. Each successively higher layer of cache may be larger than the lower level caches but they can be slower than the lower level caches as well. Thus, a Layer 0 cache provided closest to the core may be very small (say, storing 8 kilobytes of data) but able to retrieve and furnish data at clock speeds that rival the faster clock speeds of the core itself. Increasingly higher level caches, layer 1 and layer 2 caches 620, 630, may store greater amounts of data (640 kilobytes and 2 megabytes) but must operate at relatively slower clock speeds.

[43]   During operation, when a load instruction is being executed, a load request first is applied to cache layer 0. If the request hits the layer 0 cache, it furnishes the data. Otherwise, if the request misses the layer 0 cache, the request is applied to the level 1 cache. Each level of cache performs a cache lookup (determines a hit or a miss) and furnishes the data if it is able to do so. Otherwise, the request advances to successively

higher levels of cache until it is determined that no cache stores the requested data. In this event, the data is requested from system memory (represented as 640 in FIG. 8). In this regard, the architecture and operation of cache hierarchies is well known.

[44]   According to an embodiment, one or more of the layers of cache may have an architecture according to the parallel cachelet scheme. Cache layer 0 610 is shown as comprising parallel cachelets. In this case, when data requests are applied to cache layer 0 610, each request may be assigned and applied to a respective cachelet. If a request misses the cachelet, it may be applied to the layer 1 cache 620. The layer 1 cache 620 may be (but need not be) provisioned according to the parallel cachelet architecture.

[45]   Store instructions may be treated differently from loads. In an embodiment, the parallel cachelets may operate according to a write-through and write-no-allocate policy. When a store retires, its data may be broadcast to all cachelets in a layer to update them. Each cachelet may perform an independent tag match. Those cachelets that register a tag match may store the newly broadcast data in the corresponding set. Retiring stores interfere with overall performance only when store retirement is holding up the pipeline.

[46]   Several embodiments of the present invention are specifically illustrated and described herein. However, it will be appreciated that modifications and variations of the present invention are covered by the above teachings and within the purview of the appended claims without departing from the spirit and intended scope of the invention.